QT

Team Emertxe

Graphics View

Objectives

- Using GraphicsView Classes
- Coordinate Systems and Transformations
- Widgets in a Scene
- Drag and Drop
- Effects
- Performance Tuning

Objectives

- Using QGraphicsView-related classes
- Coordinate Schemes, Transformations
- Extending items
  - Event handling
  - Painting
  - Boundaries

Graphics View

- Using GraphicsView Classes
- Coordinate Systems and Transformations
- Widgets in a Scene
- Drag and Drop
- Effects
- Performance Tuning

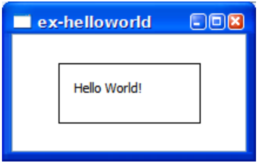
GraphicsView Framework

- Provides:
  - A surface for managing interactive 2D graphical items
  - A view widget for visualizing the items
- Uses MVC paradigm
- Resolution Independent
- Animation Support
- Fast item discovery, hit tests, collision detection
  - Using Binary Space Paritioning (BSP) tree indexes
- Can manage large numbers of items (tens of thousands)
- Supports zooming, printing and rendering

Hello World

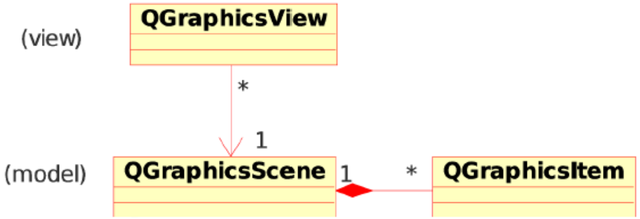
```
#include <QtWidgets>
int main(int argc, char **argv) {
    QApplication app(argc, argv);
    QGraphicsView view;
    QGraphicsScene *scene = new QGraphicsScene(&view);
    view.setScene(scene);
    QGraphicsRectItem *rect =
        new QGraphicsRectItem(-10, -10, 120, 50);
    scene->addItem(rect);
    QGraphicsTextItem *text = scene->addText( "Hello
    World!");
    view.show();
    return app.exec();
}
```

Demo



UML relationship

- QGraphicsScene is:
  - a "model" for QGraphicsView
  - a "container" for QGraphicsItems



QGraphicsScene

- Container for Graphic Items
  - Items can exist in only one scene at a time
- Propagates events to items
  - Manages Collision Detection
  - Supports fast item indexing
  - Manages item selection and focus
- Renders scene onto view
  - z-order determines which items show up in front of others

QGraphicsScene

- `addItem()`
  - Add an item to the scene
    - (remove from previous scene if necessary)
  - Also `addEllipse(), addPolygon(), addText(), etc`

```
QGraphicsEllipseItem *ellipse =
scene->addEllipse(-10, -10, 120, 50);
QGraphicsTextItem *text =
scene->addText("Hello World!");
```

- `items()`
  - returns items intersecting a particular point or region
- `selectedItems()`
  - returns list of selected items
- `sceneRect()`
  - bounding rectangle for the entire scene

QGraphicsView

- Scrollable widget viewport onto the scene
  - Zooming, rotation, and other transformations
  - Translates input events (from the View) into `QGraphicsSceneEvents`
  - Maps coordinates between scene and viewport
  - Provides "level of detail" information to items
  - Supports OpenGL

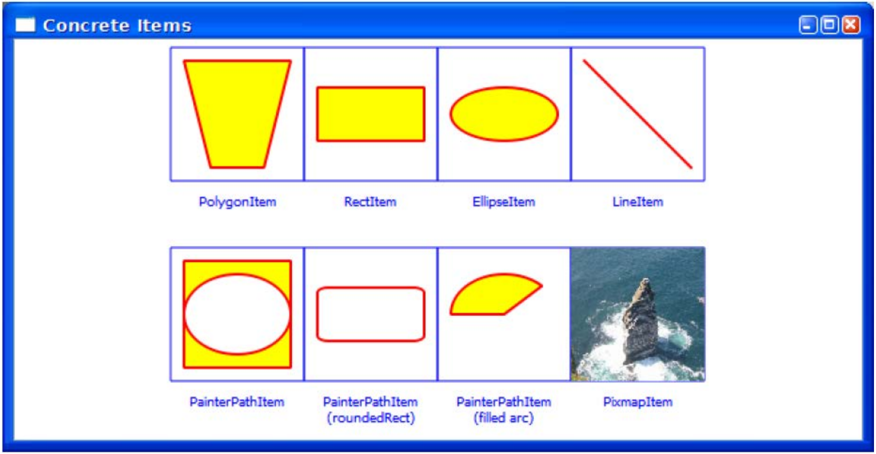
QGraphicsView

- setScene()
  - sets the QGraphicsScene to use
- setRenderHints()
  - antialiasing, smooth pixmap transformations, etc
- centerOn()
  - takes a QPoint or a QGraphicsItem as argument
  - ensures point/item is centered in View
- mapFromScene(), mapToScene()
  - map to/from scene coordinates
- scale(), rotate(), translate(), matrix()
  - transformations

QGraphicsItem

- Abstract base class: basic canvas element
    - Supports parent/child hierarchy
- Easy to extend or customize concrete items:
    - `QGraphicsRectItem`, `QGraphicsPolygonItem`, `QGraphicsPixmapItem`, `QGraphicsTextItem`, etc.
    - SVG Drawings, other widgets
- Items can be transformed:
    - move, scale, rotate
    - using local coordinate systems
- Supports Drag and Drop similar to QWidget

QGraphicsItem
Types



Demo

QGraphicsItem methods

- `pos()`
  - get the item's position in scene
- `moveBy()`
  - Moves an item relative to its own position.
- `zValue()`
  - get a Z order for item in scene
- `show()`, `hide()` - set visibility
- `setEnabled(bool)` - disabled items can not take focus or receive events
- `setFocus(Qt::FocusReason)` - sets input focus.
- `setSelected(bool)`
  - select/deselect an item
  - typically called from `QGraphicsScene::setSelectionArea()`

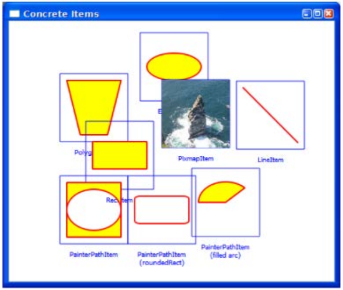
Select, Focus, Move

- QGraphicsItem::setFlags()
  - Determines which operations are supported on an item
- QGraphicsItemFlags
  - QGraphicsItem::ItemIsMovable
  - QGraphicsItem::ItemIsSelectable
  - QGraphicsItem::ItemIsFocusable

```
item->setFlags(
QGraphicsItem::ItemIsMovable|QGraphicsIt
em::ItemIsSelectable);
```

Groups of Items

- Any QGraphicsItem can have children
- QGraphicsItemGroup is an invisible item for grouping child items
- To group child items in a box with an outline (for example), use a QGraphicsRectItem



Parents and Children

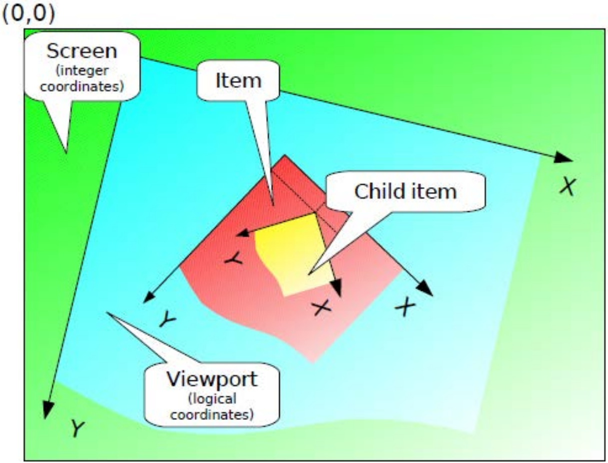
- Parent propagates values to child items:
  - `setEnabled()`
  - `setFlags()`
  - `setPos()`
  - `setOpacity()`
  - `etc...`
- Enables composition of items.

Graphics View

- Using GraphicsView Classes
- Coordinate Systems and Transformations
- Widgets in a Scene
- Drag and Drop
- Effects
- Performance Tuning

Coordinate Systems

Each View and Item has its own local coordinate system



Coordinates

- Coordinates are local to an item
  - Logical coordinates, not pixels
  - Floating point, not integer
  - Without transformations, 1 logical coordinate = 1 pixel.
- Items inherit position and transform from parent
- zValue is relative to parent
- Item transformation does not affect its local coordinate system
- Items are painted recursively
  - From parent to children
  - in increasing zValue order

QTransform

Coordinate systems can be transformed using QTransform

QTransform is a 3x3 matrix describing a linear transformation from (x,y) to (xt, yt)

m11	m12	m13
m21	m22	m23
m31	m32	m33

```
xt = m11*x + m21*y + m31
yt = m22*y + m12*x + m32
if projected:
wt = m13*x + m23*y + m33
xt /= wt
yt /= wt
```

- m13 and m23
  - Control perspective transformations
- Documentation

Common Transformations

- Commonly-used convenience functions:
  - `scale()`
  - `rotate()`
  - `shear()`
  - `translate()`
- Saves you the trouble of defining transformation matrices
- rotate() takes optional 2nd argument: axis of rotation.
  - Z axis is "simple 2D rotation"
  - Non-Z axis rotations are "perspective" projections.

View transformations

```
t = QTransform();         // identity matrix
t.rotate(45, Qt::ZAxis); // simple rotate
t.scale(1.5, 1.5)         // scale by 150%
view->setTransform(t);    // apply transform to
                            entire view
```

- `setTransformationAnchor()`
  - An anchor is a point that remains fixed before/after the transform.
  - `AnchorViewCenter: (Default)` The center point remains the same
  - `AnchorUnderMouse:` The point under the mouse remains the same
  - `NoAnchor:` Scrollbars remain unchanged.

Item Transformations

- `QGraphicsItem` supports same transform operations:
  - `setTransform(), transform()`
  - `rotate(), scale(), shear(), translate()`

.

An item's effective transformation:
The product of its own and all its ancestors' transformations

TIP: When managing the transformation of items, store the desired rotation, scaling etc. in member variables and build a QTransform from the identity transformation when they change. Don't try to deduce values from the current transformation and/or try to use it as the base for further changes.

Zooming

- Zooming is done with view->scale()

```
void MyView::zoom(double factor)
{
   double width =
   matrix().mapRect(QRectF(0, 0, 1,
   1)).width();
   width *= factor;
   if ((width < 0.05) || (width > 10))
   return;
   scale(factor, factor);
}
```

Ignoring Transformations

- Sometimes we don't want particular items to be transformed before display.
- View transformation can be disabled for individual items.
- Used for text labels in a graph that should not change size when the graph is zoomed.

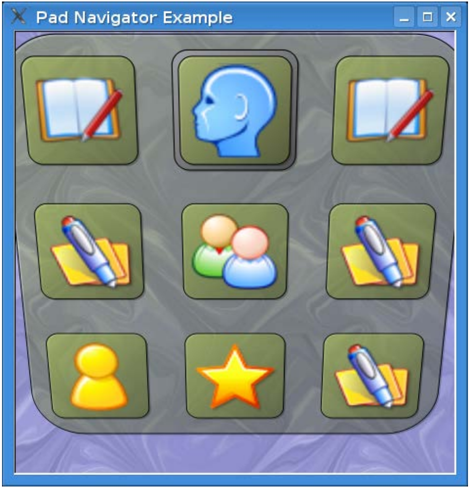
```
item->setFlag(
QGraphicsItem::ItemIgnoresTransformations);
```

Demo

Graphics View

- Using GraphicsView Classes
- Coordinate Systems and Transformations
- Widgets in a Scene
- Drag and Drop
- Effects
- Performance Tuning

Widgets in a Scene



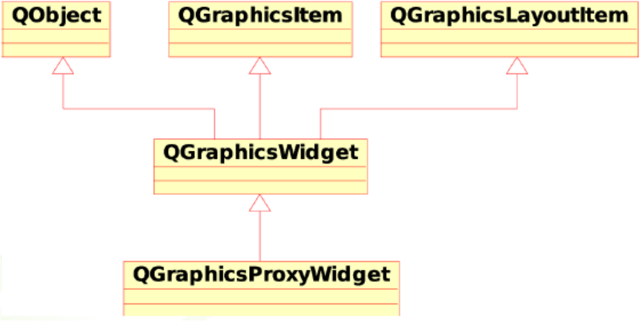
Demo

Items
not widgets

- QGraphicsItem:
  - Lightweight compared to QWidget
  - No signals/slots/properties
  - Scenes can easily contain thousands of Items
  - Uses different QEvent sub-hierarchy (derived from
  - QGraphicsSceneEvent)
  - Supports transformations directly
- QWidget:
  - Derived from QObject (less light-weight)
  - supports signals, slots, properties, etc
  - can be embedded in a QGraphicsScene with a QGraphicsProxyWidget

QGraphicsWidget

- Advanced functionality graphics item
- Provides signals/slots, layouts, geometry, palette, etc.
- Not a QWidget!
- Base class for QGraphicsProxyWidget



QObject

QGraphicsItem

QGraphicsLayoutItem

QGraphicsWidget

QGraphicsProxyWidget

QGraphicsProxyWidget

- QGraphicsItem that can embed a QWidget in a QGraphicsScene
- Handles complex widgets like QFileDialog
- Takes ownership of related widget
  - Synchronizes states/properties:
    - visible, enabled, geometry, style, palette, font, cursor, sizeHint,windowTitle, etc
    - Proxies events between Widget and GraphicsView
  - If either (widget or proxy) is deleted, the other is also!
- Widget must not already have a parent
  - Only top-level widgets can be added to a scene

Embedded Widget

```
#include <QtWidgets>
int main(int argc, char **argv) {
   QApplication app(argc, argv);
   QCalendarWidget *calendar = new
   QCalendarWidget;
   QGraphicsScene scene;
   QGraphicsProxyWidget *proxy =
   scene.addWidget(calendar);
   QGraphicsView view(&scene);
   view.show();
   return app.exec();
}
```

QGraphicsLayout

- For layout of QGraphicsLayoutItem (+derived) classes in QGraphicsView
- Concrete classes:
  - QGraphicsLinearLayout: equivalent to QBoxLayout, arranges items horizontally or vertically
  - QGraphicsGridLayout: equivalent to QGridLayout, arranges items in a grid
- QGraphicsWidget::setLayout() - set layout for child items of this QGraphicsWidget

Graphics View

- Using GraphicsView Classes
- Coordinate Systems and Transformations
- Widgets in a Scene
- Drag and Drop
- Effects
- Performance Tuning

Drag and Drop

- Items can be:
  - Dragged
  - Dropped onto other items
  - Dropped onto scenes
    - for handling empty drop areas

Start Drag

Starting an item drag is similar to dragging from a QWidget.

- Override event handlers:
  - `mousePressEvent()`
  - `mouseMoveEvent()`
- In `mouseMoveEvent()`, decide if drag started? if so:
  - Create a `QDrag` instance
  - Attach a `QMimeData` to it
  - Call `QDrag::exec()`
    - Function returns when user drops
    - Does not block event loop
- Demo

Drop on a scene

- Override `QGraphicsScene::dropEvent()`
  - To accept drop:
    - `acceptProposedAction()`
    - `setDropAction(Qt::DropAction); accept();`
- Override `QGraphicsScene::dragMoveEvent()`
- Optional overrides:
  - `dragEnterEvent(), dragLeaveEvent()`

Hands-on

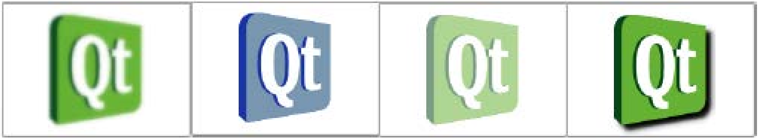
Graphics View

- Using GraphicsView Classes
- Coordinate Systems and Transformations
- Widgets in a Scene
- Drag and Drop
- Effects
- Performance Tuning

Graphics Effects

Effects can be applied to graphics items:

- Base class for effects is QGraphicsEffect.
- Standard effects include blur, colorize, opacity and drop shadow.
- Effects are set on items.
  - `QGraphicsItem::setGraphicsEffect()`
- Effects cannot be shared or layered.
- Custom effects can be written.



Graphics Effects

- Applying a blur effect to a pixmap.

```
QPixmap pixmap(":/images/qt-banner.png");
QGraphicsItem *blurItem = scene->
addPixmap(pixmap);
QGraphicsBlurEffect *blurEffect = new
QGraphicsBlurEffect();
blurItem->setGraphicsEffect(blurEffect);
blurEffect->setBlurRadius(5);
```

- An effect is owned by the item that uses it.
- Updating an effect causes the item to be updated.

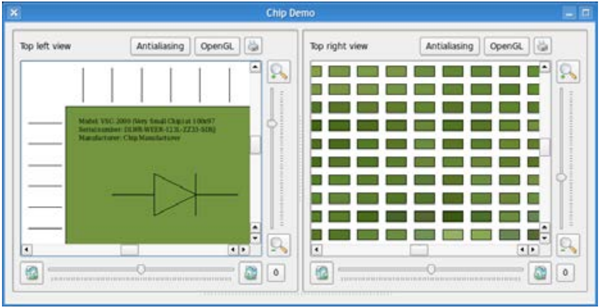
Graphics View

- Using GraphicsView Classes
- Coordinate Systems and Transformations
- Widgets in a Scene
- Drag and Drop
- Effects
- Performance Tuning

Level of Detail

- Don't draw what you can't see!
- `QStyleOptionGraphicsItem` passed to `paint()`
  - Contains palette, state, matrix members
  - `qreal levelOfDetailFromTransform(QTransform T)` method
- "levelOfDetail" is max width/height of the unity rectangle needed to draw this shape onto a QPainter with a QTransform of T.
- use `worldTransform()` of painter for current transform.
  - Zoomed out: `levelOfDetail` < 1.0
  - Zoomed in: `levelOfDetail` > 1.0

Examples



Model: VSC-2000 (Very Small Chip) at 100x97
Serial number: DLWR-WEER-123L-ZZ33-SDSJ
Manufacturer: Chip Manufacturer

Demo

Caching tips

- Cache item painting into a pixmap
  - So `paint()` runs faster
- Cache `boundingRect()` and `shape()`
  - Avoid recomputing expensive operations that stay the same
  - Be sure to invalidate manually cached items after zooming and other transforms

```
QRectF MyItem::boundingRect() const {
if (m_rect.isNull()) calculateBoundingRect();
return m_rect;
}
QPainterPath MyItem::shape() const {
if (m_shape.isEmpty()) calculateShape();
return m_shape;
}
```

setCacheMode()

- Property of `QGraphicsView` and `QGraphicsItem`
- Allows caching of pre-rendered content in a `QPixmap`
  - Drawn on the viewport
  - Especially useful for gradient shape backgrounds
  - Invalidated whenever view is transformed.

```
QGraphicsView view;
view.setBackgroundBrush(QImage(":/images/backgroundtile.png"));
view.setCacheMode(QGraphicsView::CacheBackground);
```

Tweaking

The following methods allow you to tweak performance of view/scene/items:

- `QGraphicsView::setViewportUpdateMode()`
- `QGraphicsView::setOptimizationFlags()`
- `QGraphicsScene::setItemIndexMethod()`
- `QGraphicsScene::setBspTreeDepth()`
- `QGraphicsItem::setFlags()`
    - `ItemDoesntPropagateOpacityToChildren` and `ItemIgnoresParentOpacity` especially recommended if your items are opaque!

Tips for better performance

- `boundingRect()` and `shape()` are called frequently so they should run fast!
  - `boundingRect()` should be as small as possible
  - `shape()` should return simplest reasonable path
- Try to avoid drawing gradients on the painter. Consider using pre-rendered backgrounds from images instead.
- It is costly to dynamically insert/remove items from the scene. Consider hiding and reusing items instead.
- Embedded widgets in a scene is costly.
- Try using a different paint engine (OpenGL, Direct3D, etc)
  - `setViewport (new QGLWidget);`
- Avoid curved and dashed lines
- Alpha blending and antialiasing are expensive

Hand-on

Qt Multithreading

Multithreading

- Most GUI applications have a single thread of execution in which the event loop is running
- However, if the user invokes a time consuming operation the interface freezes. We can work around this in different ways:
  - Using the `QApplication::processEvent()` during long tasks to make sure events (key, window, etc.) are delivered and the UI stays responsive.
  - Using threads to perform the long running tasks. Qt has a number of options for this.

Multithreading

Technologies

- QThread: Low-Level API with Optional Event Loops
- QThreadPool and QRunnable: Reusing Threads
- QtConcurrent: Using a High-level API
- WorkerScript: Threading in QML

QThread

- QThread is the central class in Qt to run code in a different thread
- It's a QObject subclass
  - Not copiable/moveable
  - Has signals to notify when the thread starts/finishes
- It is meant to manage a thread

QThread usage

- To create a new thread executing some code, subclass **QThread** and reimplement `run()`
- Then create an instance of the subclass and call `start()`
- Threads have priorities that you can specify as an optional parameter to `start(),` or change with `setPriority()`
- The thread will stop running when (some time after) returning from run()
- `QThread::isRunning()` and `QThread::isFinished()` provide information about the execution of the thread
- You can also connect to the `QThread::started()` and `QThread::finished()` signals
- A thread can stop its execution temporarily by calling one of the `QThread::sleep()` functions
    - Generally a *bad idea*, being event driven (or polling) is much much Better
- You can wait for a QThread to finish by calling wait() on it
    - Optionally passing a maximum number of milliseconds to wait

QThread caveats

From a non-main thread you cannot:

- Perform any GUI operation
  - Including, but not limited to: using any QWidget / Qt Quick / Qpixmap APIs
  - Using QImage, QPainter, etc. (i.e. "client side") is OK
  - Using OpenGL may be OK: check at runtime QOpenGLContext::supportsThreadedOpenGL()
- Call Q(Core|Gui)Application::exec()
- Be sure to always destroy all the QObjects living in secondary threads before destroying the corresponding QThread object
- Do not *ever* block the GUI thread

QThread usage

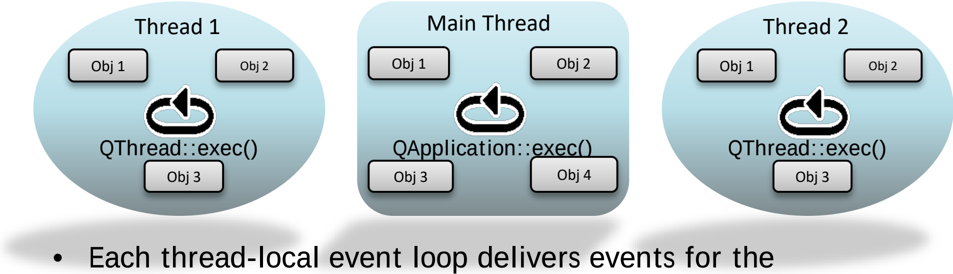
- There are two basic strategies of running code in a separate thread with QThread:
  - Without an event loop
  - With an event loop

QThread usage
without an event loop

- Subclass QThread and override `QThread::run()`
- Create an instance and start the new thread via `QThread::start()`
- Demo

QThread usage with an event loop

- An event loop is necessary when dealing with timers, networking, queued connections, and so on.
- Qt supports per-thread event loops:



- Each thread-local event loop delivers events for the QObjects living in that thread.

QThread usage with an event loop

- We can start a thread-local event loop by calling QThread::exec() from within run():

```
class MyThread : public QThread {
private:
 void run() override {
     auto socket = new QTcpSocket;
     socket->connectToHost(...);
     exec(); // run the event loop
     // cleanup
     }
};
```

Demo

- QThread::quit() or QThread::exit() will quit the event loop
- We can also use QEventLoop
  - Or manual calls to QCoreApplication::processEvents()
- The default implementation of QThread::run() actually calls QThread::exec()
- This allows us to run code in other threads without sub classing QThread:

QtConcurrent

- QtConcurrent is a namespace that provides higher-level classes and algorithms for writing concurrent software.
- Using QtConcurrent's functional map/filter/reduce algorithms, which apply functions in parallel to each item in a container.
- You can write a program that automatically takes advantage of the system's multiple cores by distributing the processing across the threads managed by the thread pool.

QtConcurrent

- Qt Concurrent supports several STL-compatible container and iterator types, but works best with Qt containers that have random-access iterators, such as QList or Qvector
- Demo

QThreadPool and QRunnable

- Creating and destroying threads frequently can be expensive.
- To avoid the cost of thread creation, a thread pool can be used.
- A thread pool is a place where threads can be parked and fetched.
- We derive a class from QRunnable. The code we want to run in another thread needs to be placed in the reimplemented QRunnable::run() method.
- Demo

Synchronization

Synchronization

- Any concurrent access to shared resources must not result in a data race
- Two conditions for this to happen:
  1. At least one of the accesses is a write
  2. The accesses are not atomic and no access happens before the other

Synchronization

Qt has a complete set of cross-platform, low-level APIs for dealing with synchronization:

  - QMutex is a mutex class (recursive and non-recursive)
  - QSemaphore is a semaphore
  - QWaitCondition is a condition variable
  - QReadWriteLock is a shared mutex
  - QAtomicInt is an atomic int
  - QAtomicPointer<T> is an atomic pointer to T
- Demo’

Thread safety in Qt

A function is:

- **Thread safe:** if it's safe for it to be invoked at the same time, from multiple threads, on the same data, without synchronization
- **Reentrant:** if it's safe for it to be invoked at the same time, from multiple threads, on different data; otherwise it requires external synchronization
- **Non-reentrant (thread unsafe):** if it cannot be invoked from more than one thread at all

For classes, the above definitions apply to non-static member functions when invoked on the same instance.

Examples

- Thread safe:
  - QMutex
  - QObject::connect()
  - QCoreApplication::postEvent()
- Reentrant:
  - QString
  - QVector
  - QImage
  - value classes in general
- Non-reentrant:
  - QWidget (including all of its subclasses)
  - QQuickItem
  - QPixmap
  - in general, GUI classes are usable only from the main thread

Intriduction to QML

What is QML

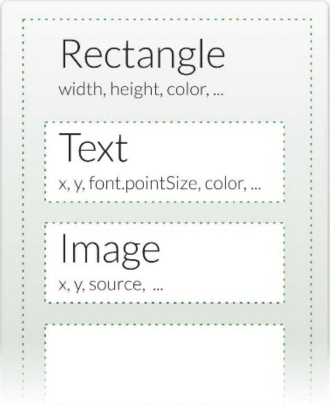
Declarative language for User Interface elements:

- Describes the user interface
  - What elements look like
  - How elements behave
- UI specified as tree of elements with properties

Elements

- Elements are structures in the markup language
  - Represent visual and non-visual parts
- Item is the base type of visual elements
  - Not visible itself
  - Has a position, dimensions
  - Usually used to group visual elements
  - Rectangle, Text, TextInput,...
- Non-visual elements:
  - States, transitions,...
  - Models, paths,...
  - Gradients, timers, etc.
- Elements contain properties
  - Can also be extended with custom properties
- QML Elements

Tree of elements



Rectangle

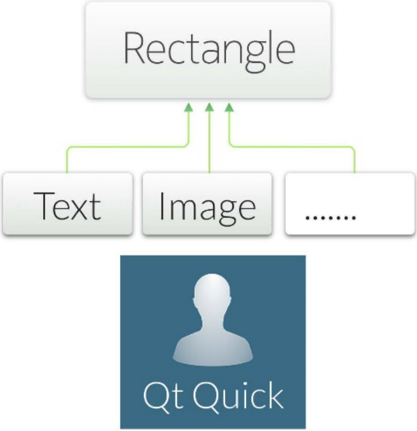
width, height, color, ...

Text

x, y, font.pointSize, color, ...

Image

x, y, source, ...



Rectangle

Text

Image

....

Qt Quick

Properties

Elements are described by properties:

- Simple name-value definitions
  - width, height, color,…
  - With default values
  - Each has a well-defined type
  - Separated by semicolons or line breaks
- Used for
  - Identifying elements (id property)
  - Customizing their appearance
  - Changing their behavior
- Demo

THANK YOU